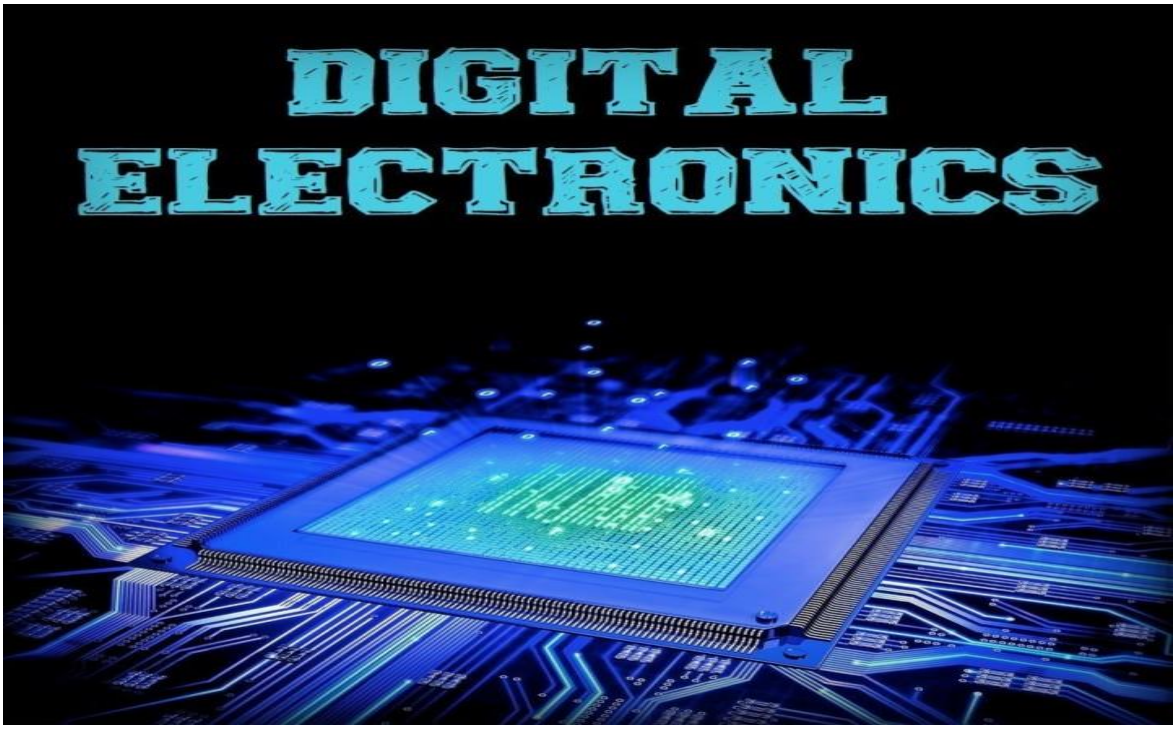
Abanob Evram

Assignmen4



1)

[Q1]

1. Implement 4-bit Ripple counter with asynchronous active low set using behavioral modelling that increment from 0 to 15
   - Input:
     - clk
     - set (sets all bits to 1)
     - out (4-bits)
2. Use the structural counter done in the previous assignment as a golden reference.
3. Test the above behavioral design using a self-checking testbench
   - Testbench should instantiate the previous two designs

The design code:

```
module Counter(clk,set,out);
input clk,set;
output reg [3:0] out;
   always @(posedge clk or negedge set) begin
       if(~set)
           out<=4'b1111;
       else
           out<=out+1;
   end
endmodule
```

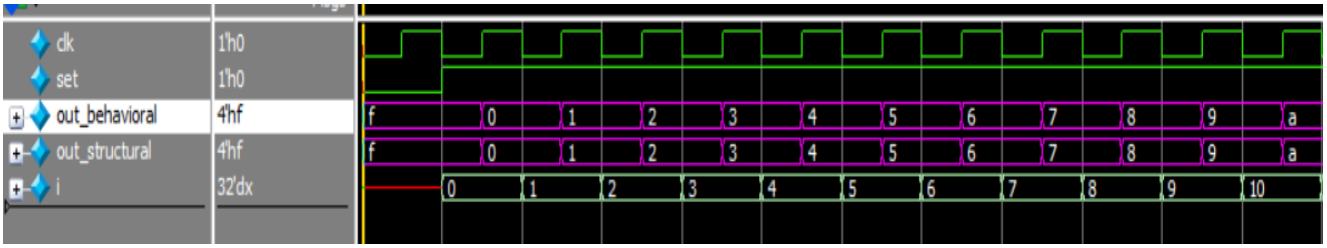
```
module D_Flipflop(d,rstn,clk,q,qpar);
input d,rstn,clk;
output reg q;
output qpar;
assign qpar = ~q ;
always @(posedge clk or negedge rstn) begin
    if (~rstn)
    q<=0;
    else
    q<=d;
end
endmodule
module Ripple_counter(clk,rstn,out);
input clk,rstn;
output [3:0] out;
wire q0,qn0,q1,qn1,q2,qn,q3,qn3;
D_Flipflop DFF0(qn0,rstn,clk,q0,qn0);
D_Flipflop DFF1(qn1,rstn,q0,q1,qn1);
D_Flipflop DFF2(qn2,rstn,q1,q2,qn2);
D_Flipflop DFF3(qn3,rstn,q2,q3,qn3);
assign out = {qn3,qn2,qn1,qn0};
endmodule
```

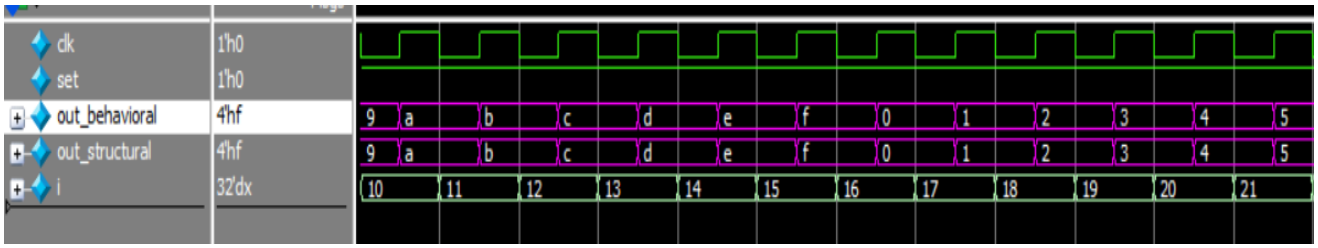
The testbench code:

```
module Counter_golden_tb();
 reg clk,set;
 wire [3:0] out_behavioral,out_structural;
 Ripple_counter golden(clk,set,out_structural);
 Counter  dut(clk,set,out_behavioral);
    initial begin
        clk=0;
        forever
        #1 clk=~clk;
        end
 integer i;
    initial begin
        set=0;
        @(negedge clk);
        if(out_behavioral!=out_structural)begin
            $display("Errrrorr");
            $stop;
        end
        set=1;
        for(i=0;i<50;i=i+1)begin
            @(negedge clk);
            if(out_behavioral!=out_structural)begin
            $display("Errrrorr");
            $stop;
            end
        end
        $stop;
    end
endmodule
```

The do file code:

```
vlib work

vlog D_Flipflop.v Ripple_counter.v Counte.v Counte_tb.v

vsim -voptargs=+acc Counter_golden_tb

add wave *

run -all

#quit -sim
```





2) Extend on the previous counter done in the previous question to have extra 2 single bit outputs (div_2 and div_4). Hint: Observe the output bits of the "out" bus to generate the following

- div_2: output signal that divides the input clock by 2
- div_4: output signal that divides the input clock by 4

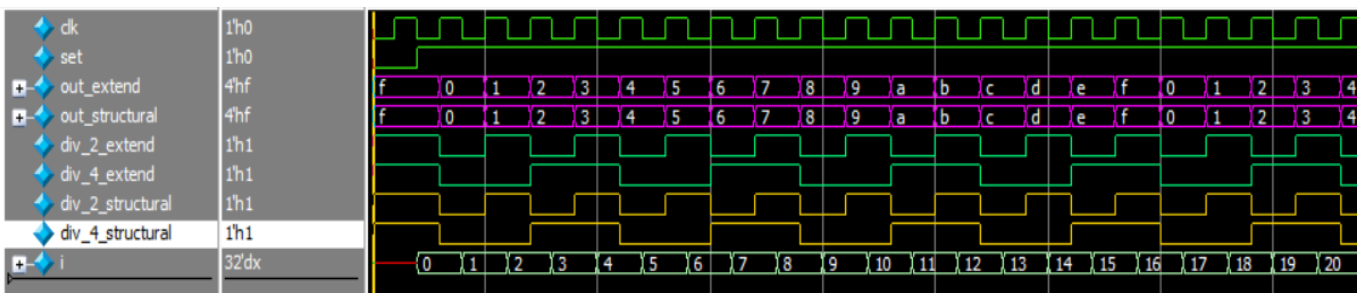
[Q2]

The design code:

```
1  module  Extend_counter(clk,set,out,div_2,div_4);
2   input clk,set;
3   output reg [3:0] out ;
4   output div_2,div_4;
5      always @(posedge clk or negedge set) begin
6          if(~set)
7              out<=4'b1111;
8          else
9              out<=out+1;
10         end
11  assign div_2 = out[0];
12  assign div_4 = out[1] ;
13 endmodule
```

The testbench code:

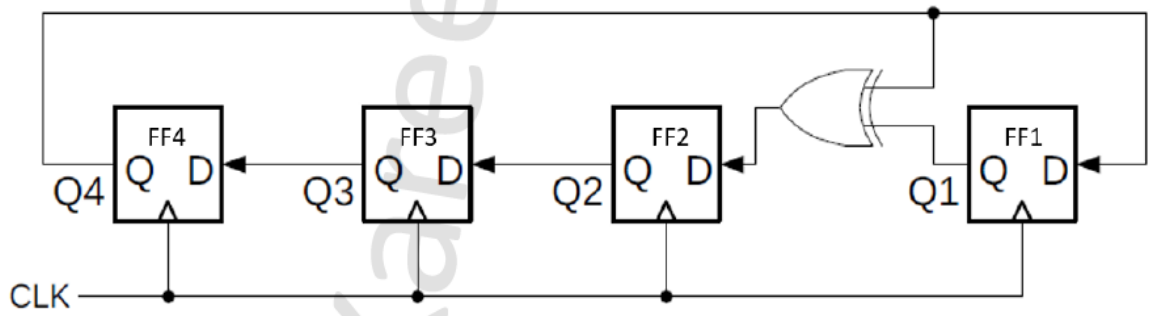
```
1  module  Extend_counter_tb();
2   reg clk,set;
3   wire [3:0] out_extend,out_structural;
4   wire div_2_extend,div_4_extend,div_2_structural,div_4_structural;
5   Ripple_counter golden(clk,set,out_structural);
6   Extend_counter  dut(clk,set,out_extend,div_2_extend,div_4_extend);
7   assign div_2_structural = out_structural[0];
8   assign div_4_structural = out_structural[1] ;
9      initial begin
10         clk=0;
11         forever
12             #1 clk=~clk;
13     end
14  integer i;
15     initial begin
16         set=0;
17         @(negedge clk);
18         if(out_extend!=out_structural) $display("Errrrorr in output");
19         if(div_2_extend!=div_2_structural) $display("Errrrorr in div_2");
20         if(div_4_extend!=div_4_structural) $display("Errrrorr in div_4");
21         set=1;
22         for (i=0;i<50;i=i+1) begin
23             @(negedge clk);
24             if(out_extend!=out_structural) $display("Errrrorr in output");
25             if(div_2_extend!=div_2_structural) $display("Errrrorr in div_2");
26             if(div_4_extend!=div_4_structural) $display("Errrrorr in div_4");
27         end
28         $stop;
29     end
30 endmodule
```



[Q3]

3) Implement the following Linear feedback shift register (LFSR)

- LFSR Inputs: clk, rst, set
- LFSR output: out (4 bits) where out[3] is connected to Q4, out[2] is connected to Q3, etc.



- LFSR can be used as a random number generator.
- The sequence is a random sequence where numbers appear in a random sequence and repeats as shown on the figure on the right

0001
0010
0100
1000
0011
0110
1100
1011
0101
1010
0111
1110
1111
1101
1001
0001

FF2, FF3, FF4 have the following specifications:

- D input
- Clk input
- Input async rst (active high) – resets output to 0
- Output Q

FF1 have the following specifications:

- D input
- Clk input
- Input async set (active high) – set output to 1
- Output Q

Note: the rst and set signals should be activated at the same time to guarantee correct operation

The design code (first design):

```
module FF1(d,clk,set,q);
input d,clk,set;
output reg q;
always @(posedge clk or posedge set) begin
    if(set)
        q<=1;
    else
        q<=d;
end
endmodule
```

```
module FF2(d,clk,rst,q);
input d,clk,rst;
output reg q;
always @(posedge clk or posedge rst) begin
    if(rst)
        q<=0;
    else
        q<=d;
end
endmodule
```

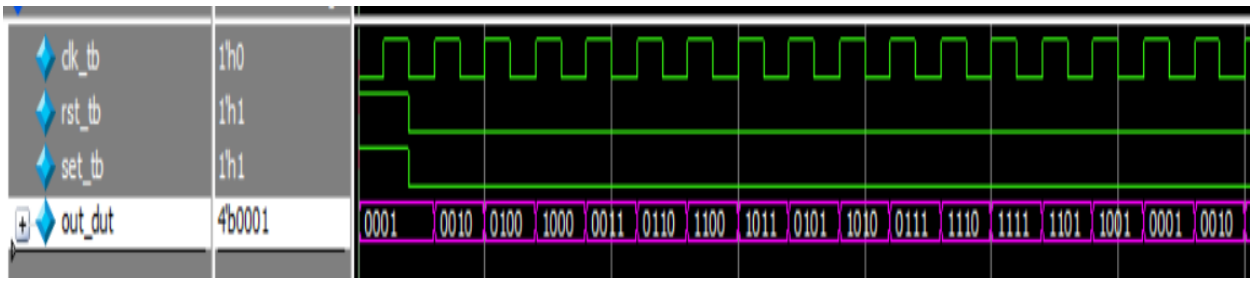
```
module LFSR(clk,rst,set,out);
 input clk,rst,set;
 output [3:0] out;
 wire q1,q2,q3,q4,d2;
 FF1   FF1(q4,clk,set,q1);
 FF2   FF2(d2,clk,rst,q2);
 FF2   FF3(q2,clk,rst,q3);
 FF2   FF4(q3,clk,rst,q4);
assign d2 =q4^q1 ;
assign out ={q4,q3,q2,q1};
endmodule
```

The design code (second design):

```
module LFSR(clk,rst,set,out);
 input clk,rst,set;
 output [3:0] out;
 reg q1,q2,q3,q4;

 always @(posedge clk or posedge set) begin
     if (set)
        q1<=1;
     else
        q1<=q4;
 end
 always @(posedge clk or posedge rst) begin
     if (rst) begin
        q2<=0;
        q3<=0;
        q4<=0;
     end
     else begin
         q2<=q4^q1;
         q3<=q2;
         q4<=q3;
     end
 end

 assign out = {q4,q3,q2,q1} ;
 endmodule
```

The testbench code:

```
1  module LFSR_tb();
2   reg clk_tb,rst_tb,set_tb;
3   wire [3:0] out_dut;
4   LFSR  dut(clk_tb,rst_tb,set_tb,out_dut);
5   initial begin
6     clk_tb=0;
7     forever
8      #1 clk_tb=~clk_tb;
9   end
10  initial begin
11     rst_tb=1;set_tb=1;
12     @(negedge clk_tb);
13     rst_tb=0;set_tb=0;
14     repeat(35)
15     @(negedge clk_tb);
16     rst_tb=1;set_tb=1;
17     @(negedge clk_tb);
18     $stop;
19  end
20 endmodule
```



[Q4]

4) Implement N-bit parameterized Full/Half adder

- Parameters
  - WIDTH: Determine the width of input a,b, sum
  - PIPELINE_ENABLE: if this parameter is high then the output of the sum and carry will be available in the positive clock edge (sequential) otherwise the circuit is pure combinational, default is high. Valid values: 0 or 1.
  - USE_FULL_ADDER: if this parameter is high then cin signal will be used during the cout and sum calculation from the input signals, otherwise if this parameter is low ignore the cin input, default is high
- Ports

Name	Type	Description
a	Input	Data input a of width determined by WIDTH parameter
b		Data input b of width determined by WIDTH parameter
clk		Clk input
cin		Carry in bit
rst		Active high synchronous reset
sum	Output	sum of a and b input of width determined by WIDTH parameter
cout		Carry out bit

The design code (first design):

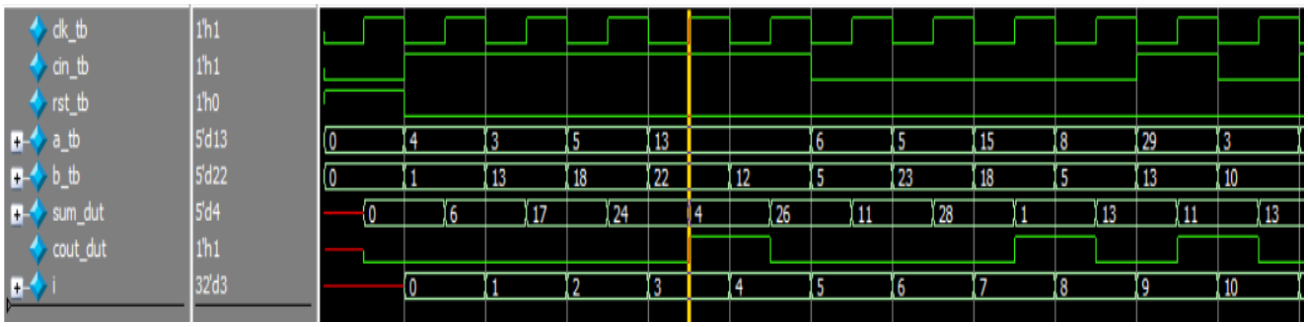
```
module Full_Half_Adder(a,b,clk,cin,rst,sum,cout);
 parameter WIDTH=5;
 parameter PIPELINE_ENABLE=1;//if high (sequential) , low (combinational)
 parameter USE_FULL_ADDER=1;//if high use cin , low ignore cin
 input [WIDTH-1:0] a,b;
 input clk,cin,rst;
 output reg cout;
 output reg [WIDTH-1:0] sum;
 reg [WIDTH:0] total ;//sum and cout
 always @(posedge clk) begin
    if(PIPELINE_ENABLE!=0) begin
       if (rst) begin
         sum<=0;
         cout<=0;
       end
       else begin
         if (USE_FULL_ADDER==0)
            total<=a+b;
         else
            total<=a+b+cin;
       end
    end
 end
 always @(*) begin
    if(PIPELINE_ENABLE==0) begin
         if (USE_FULL_ADDER==0)
            total=a+b;
         else
            total=a+b+cin;
    end
 end
 assign {cout,sum} = total ;
endmodule
```

The design code (second design):

```
module Full_Half_Adder(a,b,clk,cin,rst,sum,cout);
 parameter WIDTH=5;
 parameter PIPELINE_ENABLE=1;//if high (sequential) , low (combinational)
 parameter USE_FULL_ADDER=1;//if high use cin , low ignore cin
 input [WIDTH-1:0] a,b;
 input clk,cin,rst;
 output reg cout;
 output reg [WIDTH-1:0] sum;
 wire [WIDTH-1:0] full_sum_c,half_sum_c;
 wire full_cout_c,half_cout_c;
 reg [WIDTH-1:0] full_sum_s,half_sum_s;
 reg full_cout_s,half_cout_s;

 assign {full_cout_c,full_sum_c} = a+b+cin ;
 assign {half_cout_c,half_sum_c} = a+b ;

 always @(*) begin
    if (PIPELINE_ENABLE==0) begin
        if (USE_FULL_ADDER==0) begin
            sum = half_sum_c;
            cout = half_cout_c;
        end
        else begin
            sum = full_sum_c;
            cout = full_cout_c;
        end
    end
    else begin
        if (USE_FULL_ADDER==0) begin
            sum = half_sum_s;
            cout = half_cout_s;
        end
        else begin
            sum = full_sum_s;
            cout = full_cout_s;
        end
    end
 end

 always @(posedge clk ) begin
    if (rst) begin
        full_sum_s  <= 0;
        full_cout_s <= 0;
        half_sum_s  <= 0;
        half_cout_s <= 0;
    end
    else begin
    {full_cout_s,full_sum_s} <= a+b+cin;
    {half_cout_s,half_sum_s} <= a+b;
    end
 end
endmodule
```

The testbench code:

```
module Full_Half_Adder_tb();
 parameter WIDTH_tb=5,PIPELINE_ENABLE_tb=1,USE_FULL_ADDER=1;
 reg clk_tb,cin_tb,rst_tb;
 reg [WIDTH_tb-1:0] a_tb,b_tb;
 wire [WIDTH_tb-1:0] sum_dut;
 wire cout_dut;
 Full_Half_Adder #(WIDTH_tb,PIPELINE_ENABLE_tb,USE_FULL_ADDER) dut(a_tb,b_tb,clk_tb,cin_tb,rst_tb,sum_dut,cout_dut);
 initial begin
   clk_tb=0;
   forever
    #1 clk_tb=~clk_tb;
 end
 integer i;
 initial begin
   a_tb=0;b_tb=0;cin_tb=0;rst_tb=1;
    @(negedge clk_tb);
    rst_tb=0;
    for (i=0;i<1000;i=i+1) begin
    a_tb=$random;
    b_tb=$random;
    cin_tb=$random;
    @(negedge clk_tb);
    end
    $stop;
 end
endmodule
```



[Q5]

5) Implement shift register with the following specs:

Parameter:

1. SHIFT_DIRECTION: specify shifting direction either LEFT or RIGHT, default = “LEFT”
2. SHIFT_AMOUNT: specify the number of bits to be shifted, possible values are 1, 2, 3, 4, 5, 6, 7. Default = 1

Ports:

1. Inputs:
   - clk
   - rst (async active high)
   - load: control signal if high, register should be loaded with the input “load_value”
   - load_value: value to be loaded to the register
2. Outputs:
   - PO (8 bits): parallel out which represent the register to be shifted.

Create 2 testbench to test the operation of the register when shifting right and shift amount is 2 and the other testbench to test the shifting left and shift amount is 1. The following specs should be tested:

1. Test reset that it forces the output to zero
2. Load signal to load a randomized value to the output
3. Test the shifting operation on the output
4. Load another randomized value to the output
5. Test the shifting again

The design code:

```
module Shift_register(clk,rst,load,load_value,po);
 parameter SHIFT_DIRECTION = "LEFT" ; //LEFT OR RI-GHT
 parameter SHIFT_AMOUNT = 1; //possible values 1:7
 input clk,rst,load;
 input [7:0] load_value;
 output reg [7:0] po;
 always @(posedge clk or posedge rst) begin
    if (rst)
        po<=0;
    else if (load)
        po<=load_value;
    else begin
        if(SHIFT_DIRECTION=="RIGHT")
            po<=po>>SHIFT_AMOUNT;//another way po<={SHIFT_AMOUNT{1'b0},po[7:SHIFT_AMOUNT]}
        else
            po<=po<<SHIFT_AMOUNT;//another way po<={po[7-SHIFT_AMOUNT:0],SHIFT_AMOUNT{1'b0}}
    end
 end
endmodule
```

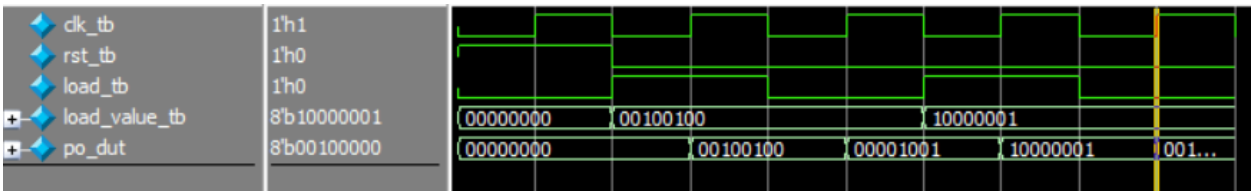
The testbench first code:

```
module Shift_tb1();
 parameter SHIFT_DIRECTION_TB = "RIGHT" ; //LEFT OR RIGHT
 parameter SHIFT_AMOUNT_TB = 2; //possible values 1:7
 reg clk_tb,rst_tb,load_tb;
 reg [7:0] load_value_tb;
 wire [7:0] po_dut;
 Shift_register #(SHIFT_DIRECTION_TB,SHIFT_AMOUNT_TB) dut( clk_tb,rst_tb,load_tb,load_value_tb,po_dut);
 initial begin
    clk_tb=0;
    forever
        #1 clk_tb=~clk_tb;
 end
 initial begin
    rst_tb=1;load_tb=0;load_value_tb=0;
    @(negedge clk_tb );
    rst_tb=0;load_tb=1;load_value_tb=$random;
    @(negedge clk_tb );
    load_tb=0;
    @(negedge clk_tb );
   load_tb=1;load_value_tb=$random;
   @(negedge clk_tb );
   load_tb=0;
   @(negedge clk_tb );
   $stop;
end
endmodule
```

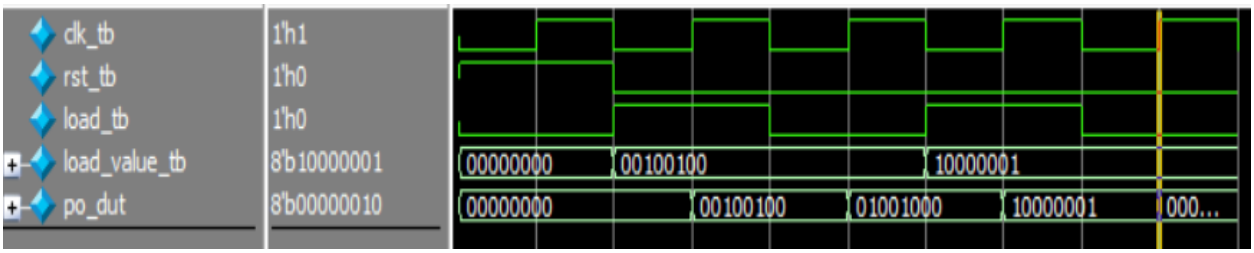
The testbench second code:

```
module Shift_tb2();
 parameter SHIFT_DIRECTION_TB = "LEFT" ; //LEFT OR RIGHT
 parameter SHIFT_AMOUNT_TB = 1; //possible values 1:7
 reg clk_tb,rst_tb,load_tb;
 reg [7:0] load_value_tb;
 wire [7:0] po_dut;
 Shift_register #(SHIFT_DIRECTION_TB,SHIFT_AMOUNT_TB) dut( clk_tb,rst_tb,load_tb,load_value_tb,po_dut);
 initial begin
    clk_tb=0;
    forever
        #1 clk_tb=~clk_tb;
 end
 initial begin
    rst_tb=1;load_tb=0;load_value_tb=0;
    @(negedge clk_tb );
    rst_tb=0;load_tb=1;load_value_tb=$random;
    @(negedge clk_tb );
    load_tb=0;
    @(negedge clk_tb );
   load_tb=1;load_value_tb=$random;
   @(negedge clk_tb );
   load_tb=0;
   @(negedge clk_tb );
   $stop;
end
endmodule
```

Wave of shifting right:



Wave of shifting left:



[Q6]

6) Implement a gray counter

Inputs:

- clk
- rst

Outputs:

- gray_out, 2-bit output

Hint: create a 2-bit binary counter counting 0, 1, 2, 3 and use the relation between the binary counter and the gray counter to assign the output bits. The most significant bit will be the same while the least significant bit of the gray out will be the reduction xor of the binary counter bits.

Create a testbench testing the following:

1. rst to force output to zero
2. remove reset and check the gray pattern from the waveform

The design code:

```
module gray_counter(clk,rst,gray_out);
 input clk,rst;
 output [1:0] gray_out;
 reg [1:0] bin_count;
 always @(posedge clk) begin
       if (rst)
        bin_count<=0;
       else
           bin_count<=bin_count+1;
 end
 assign gray_out[1] = bin_count[1];
 assign gray_out[0] = ^bin_count;
endmodule
```

The testbench first code:

```
module gray_counter_tb();
 reg clk_tb,rst_tb;
 wire [1:0] gray_out_dut;
 gray_counter dut(clk_tb,rst_tb,gray_out_dut);
 initial begin
   clk_tb=0;
   forever
       #1 clk_tb=~clk_tb;
 end
 initial begin
   rst_tb=1;
    @(negedge clk_tb);
    rst_tb=0;
    repeat(12)
    @(negedge clk_tb);
    $stop;
 end

endmodule
```